Ian Zhang
Professor Jun Yang
Computer Science 316
2 December 2013
Assignment #4

Problem 1
Gradiance

Problem 2
A table (,,...)with100,000rows is stored in2,500disk blocks. The rows are sorted by's primary key, but not by. There is a dense, secondary B+-tree index on (), which has 3 levels and 500 leaves. Suppose we want to sort by. We have 51 memory blocks at our disposal. Method 1 performs an external-memory merge sort using all memory available. Method 2 takes advantage of the fact that the values of are already sorted in the B+-tree index o (): It simply scans the leaves of the index to retrieve and output rows in order.

How many disk I/O's do these two methods require? Which one is the winner?

T(R) = 100,000
B(R) = 2,500

Method 1: External merge sort.

I/O = 3 * B(R) = 7500 Disk I/Os

2500 to bring each block into memory to read. 2500 to write and sort. 2500 for the last, sorted read through.

Method 2: B+ tree traversal

I/O = 100,000 disk I/Os to access all 200 pointers of all 500 leaves. We also need 500 disk I/Os to traverse the leaves.

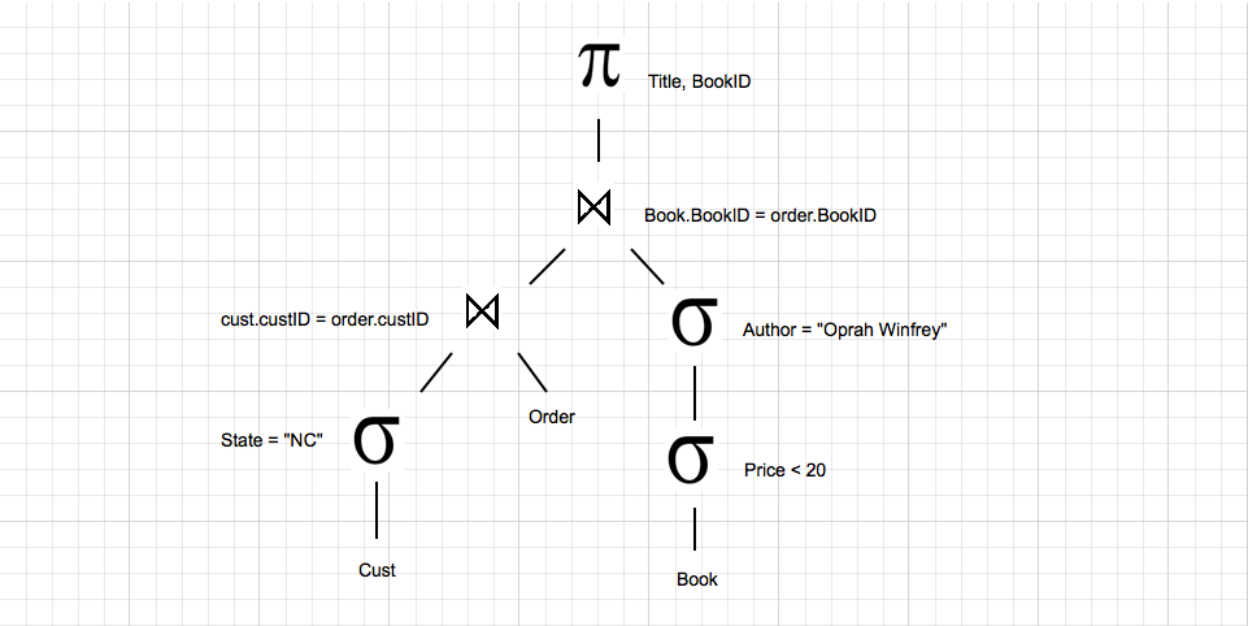
This gives us a total of 100,500 disk I/Os for method 2.

Note: This is assuming a worst case scenario where the cache replacement policy is so atrocious that it essentially misses 100% of the time when bringing disks into and reading from memory. In reality, we may be able to cache each data block in an intelligent way to reduce I/Os significantly.

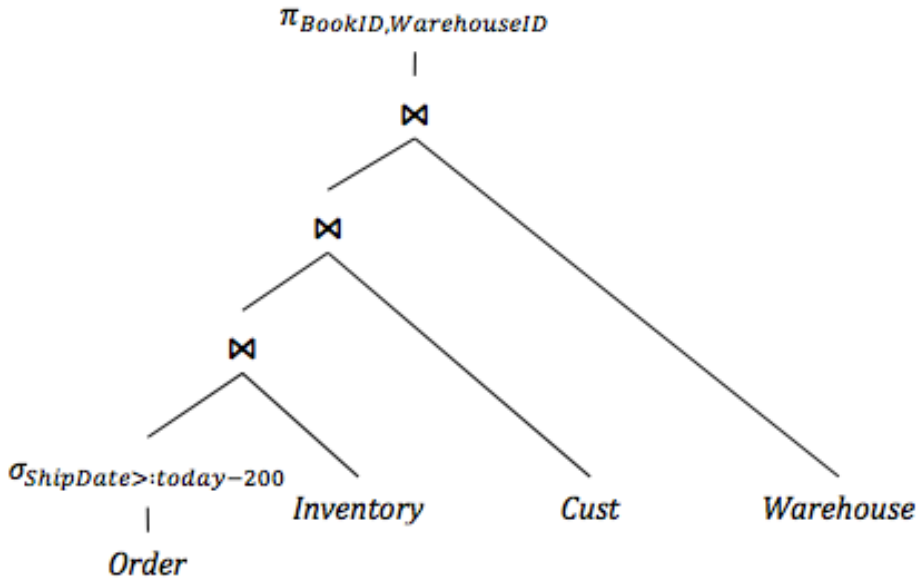
External merge sort wins this sorting battle by a large margin.

Problem 3

a.)



(b)



Goal: find book ids and how many warehouseids are associated with them

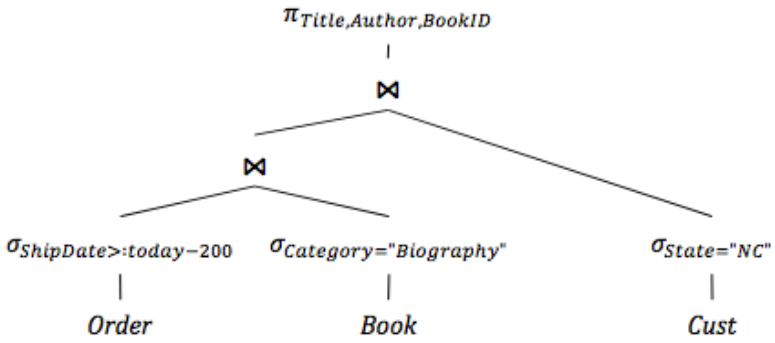
We begin with 60,000 orders, which are first filtered based on the number of days since the order. "Today" is defined as starting as 1,000 days ago (assumption from Professor Yang's piazza post). That narrows us from 1000 days – 200 days = 1/5 of the total orders.

Plugging this in to our equation for selecting tuples (|R| / max (A in R)) is 60,000 / 5 = 12k orders. We then join the number of orders with our inventory on the attribute book id to the find the unique books within those orders. There are 1,000 book ids in the inventory. Plugging this in to our equation for equijoin tuples (|R| * |S| / max (A from R, A from S)) is 12k * 40k / 1k = 480k unique orders.

Moving to the second equijoin, we join the 480k unique orders with 3k customers on the attribute customer id to find how many unique customers have placed these 480k unique orders. Again plugging this into our equijoin equation we have 480k * 3k / 3k = 480k unique orders among 3k customers.

Finally we move to the last equijoin to determine how many of the unique 50 warehouses are to house these 480k unique orders among 3k unique customers in which unique state. So we here are performing a 2 attribute equijoin, on state and on warehouse id. Plugging this once more into our equijoin we have 480k * 50 / max (50 [warehouse id] x 50 [state]), 50) = 480k / 50 = **9.6k total tuples for this relation.**

(c)



Goal: find book ids and how many authors and titles are associated with them

We begin with 60,000 orders, which are first filtered based on the number of days since the order. "Today" is defined as starting as 1,000 days ago (assumption from Professor Yang's piazza post). That narrows us from 1000 days – 200 days = 1/5 of the total orders.

Plugging this in to our equation for selecting tuples (|R| / max (A in R)) is 60,000 / 5 = 12k orders as in problem b. We then filter 1k books based on the category "biography", which I am assuming to be one of ten equally distributed categories among books. Plugging this in to our equation for selecting tuples (|R| / max (A in R)) is 1k / 10 = 100 biography books. We then equijoin those 100 biography books with the 12k orders shipped within the last 200 days on the attribute book id. Plugging this into our equijoin equation we get (12k * 100 / 1000) = 1.2k unique biography books shipped within the last 200 days.

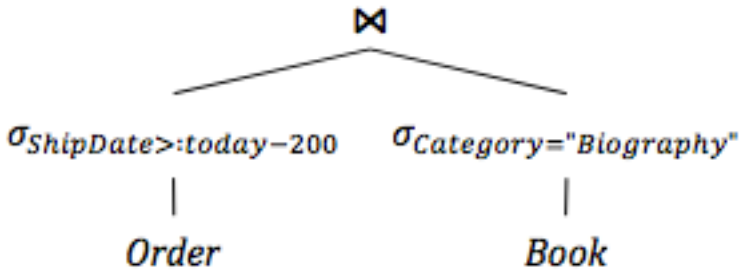
We now equijoin this 1.2k with customers residing in NC on the attribute customer id to find the unique customers in NC who ordered one of the 1.2k unique biography books shipped within the last 200 days. Plugging this in to our equation for selecting tuples (|R| / max (A in R)) is 3k / 50 = 60 customers who reside in the state of NC. Plugging this into our equijoin equation we get (1.2k * 60 / 3000) = 24 unique biography books. Assuming each book is truly unique and has unique authors and titles, we end up with **24 total tuples for this relation.**

For the following questions, further suppose that:

- Each disk/memory block can hold up to 10 rows (from any table);
- All tables are stored compactly on disk (10 rows per block);
- 8 memory blocks are available for query processing.

(d) Suppose that there are no indexes available at all, and records are stored in no particular order. What is the best execution plan (in terms of number of I/O's performed) you can come up with for the query $\sigma_{ShipDate=:today \text{ and } Category="Biography"}(Order \bowtie Book)$? Describe your plan and show the calculation of its I/O cost.

Relational algebra tree:



The filter on order will produce 12,000 tuples (see 3c for assumptions). The filter on book will produce 100 tuples (see 3c for assumptions). We will need to scan both sets then perform a nested loop join on attribute book id to reveal an optimal execution plan. The scan of each set results is as follows:

Order
60,000 tuples on initial scan [6,000 blocks]
12,000 tuples for Order:shipdate after filter [1,200 blocks to write out]
10 tuples per disk block
1,200 blocks total

1200 I/Os for table scan on filtered order tuples.

Book
1,000 tuples on initial scan [100 blocks]
100 tuples for Book:biography [10 blocks to write out]
10 tuples per disk block
10 blocks total

10 I/Os for table scan on filtered book tuples.

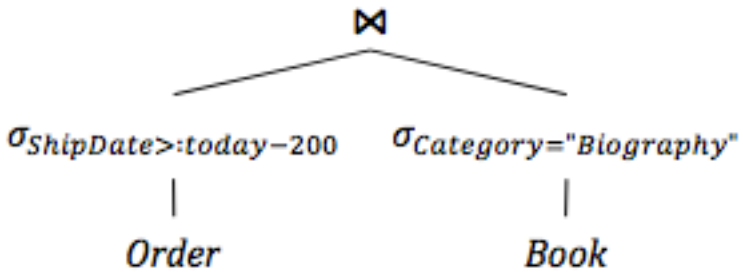
Nested loop join assuming R is relation of orders and S is relation of books:

In the worst case scenario, where R does not fit into memory, we have the following equation to express the nested loop join query where each tuple of R is evaluated against each tuple of S to check if book ids match: B(R) * B(S) / M = 1200 * 10 / 8 = 1500

Thus, all told we have 6000 + 100 + 1200 + 10 + 1500 = 8810 disk I/Os for this execution plan.

(e) Suppose a B+-tree primary index has been created for each table on its primary key, but no other indexes are available. Furthermore, assume that the B+-tree for *Order* has 4 levels with a fan-out of 10 on all levels except the root, and that the B+-tree for *Book* has 3 levels with a fan-out of 10 on all levels. What is the best plan for the same query in (d)? Again, describe your plan and show the calculation of its I/O cost.

Relational algebra tree:



In this query since we have a primary key index on some attribute a of both Order and Book, we implement a slightly different execution plan to conserve disk I/Os. First, we perform an index scan on the primary keys of both relational sets—stopping where we find our selection operators (as defined by the shipdate filter for order and biography filter for book). We then index join these two filtered sets. These operations result in the following disk I/Os:

Order:
60k tuples. B+ tree on primary index with 4 level fan-out of 10.
60,000 tuples / 10 pointers per node = 6000 leaf nodes.
60,000 tuples / 10 tuples per block = 6000 blocks.
V(R,a) = 12,000 tuples

Book:
1k tuples. B+ tree on primary index with 3 level fan-out of 10.
1,000 tuples / 10 pointers per node = 100 leaf nodes.
1,000 tuples / 10 tuples per block = 100 blocks
V(R,a) = 100 tuples

The equation for calculating an index-based selection's I/O requirements is:

B(R) / V(R,a) (where V(R,a) is the number of tuples that satisfies the selection filter) + the disk I/Os needed to access the index. Since a in both relations is the primary key, V(R,a) is in fact T(R). So for both order and book we have a total of 1 disk I/O each needed to retrieve each tuple value v that matches the selection condition a. This results in a 12000 I/O cost for Order and 100 I/O cost for Book. Finally we implement an index-based join, which entails traversing the leaves of each B+ tree to find common book id values. This will require 6000 disk I/Os + 100 disk I/Os. Thus we get a total of 12,000 + 100 + 6000 + 100 = **18,200 disk I/Os total for this execution plan.**

Problem 4
Gradiance

Extra Credit

Consider a table *R* with 100,000 rows, which are stored compactly on disk in exactly 10,000 blocks in no particular order. There are only 20,000 distinct rows in *R* (other rows are duplicates of these). We wish to compute the following query *Q*: `SELECT DISTINCT * FROM R;`

(a) What is the minimum amount of memory (in blocks) required to compute *Q* in one pass (i.e., using only 10,000 I/O's excluding the cost of writing the result)?

100,000 rows / 10,000 blocks = 10 rows per block.
A one pass algorithm resulting in 10,000 I/Os (minus the cost of writing the result) to compute 20,000 distinct rows would require 10,000 memory blocks.

In this problem we need a main memory structure that allows us to read a new tuple and to tell whether or not that tuple already exists—all in just 10,000 I/Os. Assuming a simple table-scan is used in this problem to iterate through 100,000 rows, this would incur a 10,000 I/O cost as the disk accesses for the table-scan method is 2 B(R) where B(R) is the block size. Since we do not account for writing however, we need only B(R). Thus, this results in a main memory structure with a minimum of 10,000 blocks.

(b) Suppose that we only have 1,001 blocks of memory available. Devise a strategy that can compute *Q* in no more than 20,000 I/O's in the worst case (again, excluding the cost of writing the result).

To iterate over 10,000 blocks in just 20,000 disk accesses and having 1,001 blocks of memory at our disposal, we would need to follow a strategy similar to Two Phase, Multi-way Merge Sort but modified slightly to be one that eliminates duplicates:

Phase 1: run all 10,000 blocks in disk through 1,000 memory blocks, sorting according to merge-sort (can be any algorithm for this case I just used merge-sort randomly). Write each sublist out to a secondary storage.

Phase 2: in sorted order, we repeatedly use the last block of available memory to hold one block in turn from each sorted sublist and compare it to one potential output block. We then iterate through the tuples and select the first unconsidered tuple (assuming these have been sorted correctly) among all sublists that are considered.

Phase 3: Write a copy of that tuple to the output block, and delete all occurrences of that tuple from every input block we encounter.

Following these three steps we see a three phase approach to eliminating duplicates that sorts all 10,000 blocks, and repeatedly goes through each tuple of each block and deletes the first encountered (and nicely sorted) tuple per block, then finally deletes that tuple from every other considerable input block.

This strategy will in fact delete tuples as we buffer, leading to a disk access cost of B(R) ^ ½ [on average]. This is far below the 20,000 I/O worst case scenario.